Jak przenieść kod z ARM7 do Cortex-M3?

Rdzeń Cortex-M3 cechuje się licznymi zaletami, w porównaniu do ARM7TDMI. Pozwala na szybsze wykonywanie kodu, przy mniejszym poborze mocy, a także na łatwiejsze tworzenie oprogramowania. W rezultacie świetnie nadaje się do wielu aplikacji czasu rzeczywistego i wszystko wskazuje na to, że wiele aplikacji wykonywanych obecnie na układach z ARM7 będzie przenoszonych na mikrokontrolery z Cortex-M3. Pytanie: jak to zrobić?

Wybieraj podobne układy

W praktyce, najważniejszym aspektem, który należy wziąć pod uwagę przy migracji z ARM7 do Cortex-M3 jest wybór takiego układu scalonego, którego interfejsy komunikacyjne są identyczne i który zawiera takie same wbudowane bloki peryferyjne. W przeciwnym razie konieczne będzie stworzenie nowych sterowników do elementów peryferyjnych, których napisanie może zająć wiele dni lub nawet tygodni.

Zastosowanie układów z identycznymi blokami peryferyjnymi pozwoli projektantowi ponownie wykorzystać zdecydowaną większość dotąd napisanych sterowników, stworzonych w języku C. Co więcej, producenci układów scalonych dostarczają odpowiednie pliki nagłówkowe, które zawierają aktualne adresy rejestrów poszczególnych bloków, dzięki czemu ewentualne zmiany w ich adresacji nie mają wpływu na pracę programisty. Wystarczy by programista załączył nowe pliki nagłówkowe do projektu i rekompilował kod.

Ze względu na fakt, że różnice pomiędzy omawianymi rdzeniami są jednak dosyć duże, kilka z ich cech wymaga szczególnej uwagi programisty. Poniżej prezentujemy listę tematów, które trzeba rozważyć przy przenoszeniu kodu z układów z ARM7TMDI na Cortex-M3:

- Nowy format tablicy wektorów obsługi wyjątków
- Nowy sposób wstępnej konfiguracji kodu i stosów
- Mapowanie pamięci RAM
- Nowa konfiguracja przerwań sprzętowych
- Przerwania programowe
- Procedury obsługi błędów
- Brak instrukcji SWP
- Kontrola czasu wykonywania
- Tłumaczenie wszelkiego ręcznie stworzonego kodu asemblera
- Optymalizacja

Tablica wektorów obsługi wyjątków

Tablica ta zawiera adresy tych fragmentów kodu programu, które służą do obsługi różnego rodzaju asynchronicznych zdarzeń. W przypadku rdzeni ARM do zdarzeń tych zaliczają się m.in. zerowanie (zarówno podczas uruchamiania urządzenia, jak i zwykły reset sprzętowy), błędy wynikające z nieprawidłowego działania szyny komunikacyjnej lub powstające w przypadku wczytania niezdefiniowanych instrukcji oraz przerwania, wyzwalane zarówno przez oprogramowanie, jak i np. układy peryferyjne zintegrowane z rdzeniem w jednej obudowie.

W przypadku rdzeni ARM7TDMI tablica obsługi zdarzeń specjalnych zazwyczaj składa się z przynajmniej sześciu instrukcji skoku (rozgałęzienia), ręcznie zakodowanych w asemblerze (listing 1).

W przypadku rdzeni Cortex-M3 tablica wektorów obsługi zdarzeń specjalnych może zostać zdefiniowana w języku C w postaci tablicy wskaźników. Pierwszy z jej wpisów zawiera adres stosu, a pozostałe to wskaźniki do różnych procedur obsługi wyjątków (listing 2).

Tryby pracy procesora

Układy z ARM7TMDI obsługują 7 trybów procesora, z których 6 ma własne stosy. Jeden z tych siedmiu trybów – tryb użytkownika, działa na niższym poziomie uprzywilejowania niż pozostałe. W przypadku układów z Cortex-M3 dostępne są tylko dwa tryby: normalny (thread) i obsługi przerwania (handler). Dodatkowo, w trybie normalnym procesor może pracować na poziomie uprzywilejowanym albo użytkownika, a ponadto może korzystać z systemowego stosu lub stosu użytkownika. W trybie obsługi przerwania procesor zawsze działa na poziomie uprzywilejowanym oraz może korzystać jedynie ze stosu systemowego.

Praca na uprzywilejowanym poziomie pozwala na dostęp do rejestrów stanu procesora. W przypadku ARM7TDMI są to rejestry CPSR i SPSR, a w Cortex-M3 tylko APSR. Co więcej, poziom ten otwiera dostęp do ewentualnych zastrzeżonych obszarów pamięci zdefiniowanych w jednostce ochrony pamięci (MPU – Memory Protection Unit). Różnice pomiędzy trybami pracy obu rdzeni zostały przedstawione w tabeli 1.

Konfiguracja stosów

Praktycznie wszystkie, jedynie poza najprostszymi systemami z rdzeniami ARM7TDMI używają przynajmniej dwóch trybów procesora: SVC do inicjalizacji oraz do obsługi kodu głównej pętli programu oraz IRQ do obsługi przerwań. Każdy z tych trybów musi mieć własny, zainicjalizowany stos. W tym celu konieczne jest wykonanie kodu przedstawionego na listingu 3.

Listing 1. Tablica wektorów obsługi zdarzeń specjalnych rdzenia ARM7TMDI. Zamiast ostatniego ze skoków możliwe jest bezpośrednie rozpoczęcie pod adresem 0x1C procedury obsługi szybkiego przerwania

```
0x00     Reset_Handler
0x04     UndefInstr_Handler
0x08     SWI_Handler
0x0C     PrefetchAbort_Handler
0x10     DataAbort_Handler
0x14     .                ; wektor zarezerwowany
0x18     IRQ_Handler
0x1C     FIQ_Handler
```

Listing 2. Tablica wektorów obsługi zdarzeń specjalnych rdzenia Cortex-M3 może być napisana w języku C

```
#define TOP_STACK   0x20001000
void *vector_table[] = {
    TOP_STACK,
    System_Init,
    NMI_Handler,
    HardFault_Handler,
    MemManage_Handler,
    BusFault_Handler,
    UsageFault_Handler,
    /*
     * pozostałe wskaźniki obsługi (w tym wskaźniki obsługi
przerwań pochodzących od bloków peryferyjnych)
     */
};
```

Tabela 1. Tryby pracy układów z rdzeniem ARM7TDMI i odpowiadające im tryby w Cortex-M3

ARM7TDMI				Odpowiedniki w Cortex-M3			
Wyjątek	Tryb	Dostępny stos	Poziom uprzywilejowania	Wyjątek	Tryb	Dostępny stos	Poziom uprzywilejowania
-	User	usr	Normalny	-	Normalny	Użytkownika	Normalny
-	System	usr	Wysoki	-	Normalny	Użytkownika	Wysoki
FIQ	FIQ	fiq	Wysoki	-	-	-	-
IRQ	IRQ	irq	Wysoki	IRQ	Przerwania	Systemowy	Wysoki
Reset	Supervisor	svc	Wysoki	Reset	Normalny	Systemowy	Wysoki
Software Interrupt	Supervisor	svc	Wysoki	Software Interrupt	Przerwania	Systemowy	Wysoki
Undefined Instruction	Undefined	undef	Wysoki	MemManage or Bus Fault	Przerwania	Systemowy	Wysoki
Prefetch or Data Abort	Abort	abt	Wysoki	Usage Fault	Przerwania	Systemowy	Wysoki

Listing 3. Kod asemblera potrzebny do inicjalizacji dwóch ze stosów procesorów ARM7TDMI

```
Reset_Handler:
msr CPSR_c, #ARM_MODE_IRQ | IRQ_DISABLE | FIQ_DISABLE
    ldr sp, =IRQ_STACK_START    ; ustaw wskaźnik stosu IRQ
msr CPSR_c, #ARM_MODE_SVC | IRQ_DISABLE | FIQ_DISABLE
    ldr sp, =SVC_STACK_START    ; ustaw wskaźnik stosu SVC
    ldr r0, =System_Init
    blx r0                      ; skocz do procedury System_Init
```

W układach z Cortex-M3, po zerowaniu procesor automatycznie ustawia wskaźnik stosu systemowego na wartość pierwszą wartość podaną w tablicy wektorów obsługi wyjątków (list. 2) po czym skacze do procedury wskazywanej przez drugi wpis tej tablicy (*System_Init*). Ponieważ stos systemowy może być używany zarówno przez procedury obsługi przerwań, jak i główny kod programu, w praktyce Cortexy-M3 mogą być zastosowane do wielu aplikacji bez potrzeby pisania kodu asemblera, którym inicjalizowane byłyby wskaźniki innych stosów.

Zagnieżdżone przerwania

Sposób obsługi przerwań, zarówno tych pochodzących z podsystemów scalonych ze rdzeniem procesora w jednej obudowie, jak i z urządzeń zewnętrznych w dużej mierze decyduje o jakości mikrokontrolera. Bardzo ważne jest, by obsługa przerwań odbywała się z możliwie niewielkimi opóźnieniami oraz by opóźnienia te były jak najbardziej stałe, a więc przewidywalne. W porównaniu do rdzenia ARM7TMDI, w Cortex-M3 wprowadzono liczne usprawnienia zarówno w długości występujących opóźnień, jak i ich rozrzutach oraz ułatwiono sposób programowania procedur obsługujących przerwania.

Ponieważ rdzeń ARM7TDMI ma jedynie dwa wejścia przystosowane do obsługi wszelkiego rodzaju przerwań (IRQ i FIQ), większość producentów układów z tymi rdzeniami stosuje dodatkowy kontroler przerwań, który multipleksuje sygnały przerwań pochodzące z wielu źródeł. W związku z tym, procedura obsługi przerwań IRQ i FIQ musi rozpoczynać się od określenia źródła, które zgłosiło przerwanie, by następnie skoczyć do odpowiedniego fragmentu kodu. Kolejne przerwanie nie może być obsłużone do momentu, gdy procedura obsługi poprzedniego nie zostanie zakończona, co często znacząco zwiększa czas oczekiwania na wykonanie przerwania oraz zmniejsza determinizm czasowy takiego systemu. Oczywistym rozwiązaniem tego problemu jest priorytetyzacja zdarzeń i pozwolenie tym o większym priorytecie wywłaszczanie mniej ważnych wątków.

Aby wprowadzić ten mechanizm w mikrokontrolerach z ARM7TDMI konieczne jest napisanie w asemblerze kodu obsługi przerwań (ISR – Interrupt Service Routine), który kolejno: zapisuje stan procesora na stosie, zmienia tryb procesora z IRQ na SVC, ponownie uaktywnia obsługę przerwań, a na końcu skacze do procedury obsługi przerwania o większym priorytecie (listing 4). Po jej zakończeniu, odtwarzany jest stan procesora z momentu wstrzymania obsługi poprzedniego przerwania. Ponieważ procedura obsługi przerwań jest wywoływana w momencie gdy procesor pracuje w trybie SVC, obsługa przerwania może być wstrzymana przez zdarzenie o wyższym priorytecie.

Tabela 2. Przykładowa konfiguracja priorytetów przerwań w układach z rdzeniem Cortex-M3

	Główny priorytet	Subpriorytet
IRQ0	1	0
IRQ1	2	0
IRQ2	2	1

W układach z rdzeniem Cortex-M3 kod przedstawiony na listingu 4 nie jest potrzebny, gdyż zaimplementowano w nich kontroler zagnieżdżonych przerwań (Nested Vectored Interrupt Controller). Jest on podobny w budowie do kontrolerów stosowanych przez producentów w układach z ARM7TDMI, ale ze względu na fakt, że został wbudowany w sam rdzeń, pozwala na wykonywanie bardziej zaawansowanych operacji. W praktyce – kod przedstawiony na listingu 4, w rdzeniu Cortex-M3 został zaimplementowany sprzętowo.

W momencie wystąpienia przerwania o wyższym priorytecie niż obecnie wykonywane, NVIC automatycznie zapisze rejestry potrzebne do wywołania funkcji zgodnej z AAPCS (Arm Architecture Procedure Call Standard) i odtworzy je gdy funkcja ta zakończy swoje działanie. Jeśli kompilator C, w którym pisana jest funkcja obsługi przerwania jest zgodny z AAPCS, jednostka NVIC będzie mogła bezpośrednio wywołać taką funkcje. Oznacza to, że w tablicy *vector_table[]* z listingu 2 mogą się znaleźć wskaźniki do funkcji napisanych w języku C.

Listing 4. Kod asemblera umożliwiający obsługę przerwań o różnych priorytetach w układach z ARM7TDMI

```
IRQ_Handler:
        sub     lr, lr, #4                  ; zmień LR_irq i zapisz je na stosie IRQ
        stmfd   sp!, {lr}
        mrs     r14, SPSR                   ; zapisz stary stan procesora i R0 na
                                            ; stosie IRQ
        stmfd   sp!, {r0, r14}
        ldr     r14, =INTERRUPT_CONTROLLER_VECTOR_REGISTER
        ldr     r0, [r14]                   ; pobierz wektor obsługi przerwania
                                            ; z kontrolera
        msr     CPSR_c, #ARM_MODE_SVC       ; przełącz do trybu SVC
        stmfd   sp!, {r1-r3, r12, lr}       ; zapisz rejestry na stosie SVC
        mov     lr, pc
        bx      r0                          ; skocz do obsługi przerwania
        ldmia   sp!, {r1-r3, r12, lr}       ; odtwórz rejestry ze stosu SVC
                                            ; wróć do trybu IRQ i zablokuj
                                            ; przyjmowanie przerwań
        msr     CPSR_c, #ARM_MODE_IRQ | IRQ_DISABLE
        ldmia   sp!, {r0, r14}              ; odtwórz R0 i SPSR_irq ze stosu IRQ
        msr     SPSR_cxsf, r14
        ldmia   sp!, {pc}^                  ; wyjdź z obsługi przerwań
```

Konfiguracja przerwań

Jednostka NVIC określa, który z wyjątków powinien zostać obsłużony w pierwszej kolejności, bazując na priorytetach ich źródeł. Wyjątki: zerowanie, przerwanie niemaskowalne (NMI) i błąd typu *Hard Fault* mają na stałe zdefiniowane priorytety, odpowiednio: pierwszy, drugi i trzeci. Pozostałe źródła przerwań mają priorytety nadawane przez użytkownika poprzez określenie dwóch wartości: głównego priorytetu (preempt priority) i subpriorytetu (subpriority). W momencie zaistnienia wyjątku o wyższym priorytecie głównym niż priorytet aktualnie wykonywanej procedury, rozpoczęta zostanie obsługa nowego zdarzenia. W przeciwnym wypadku

Tabela 3. Sposób podziału bitów priorytetu pomiędzy priorytet główny (kolumny pre) i subpriorytet (kolumny sub), w zależności od wielkości rejestru priorytetów oraz rodzaju ustawień grup priorytetów

Rozmiar rejestru priorytetów	3		4		5		6		7		8	
Ustawienia grup priorytetów	Pre	Sub	Pre	Sub	Pre	Sub	Pre	Sub	Pre	Sub	Pre	Sub
0	[7:5]	-	[7:4]	-	[7:3]	-	[7:2]	-	[7:1]	-	[7:1]	[0]
1	[7:5]	-	[7:4]	-	[7:3]	-	[7:2]	-	[7:2]	[1]	[7:2]	[1:0]
2	[7:5]	-	[7:4]	-	[7:3]	-	[7:3]	[2]	[7:3]	[2:1]	[7:3]	[2:0]
3	[7:5]	-	[7:4]	-	[7:4]	[3]	[7:4]	[3:2]	[7:4]	[3:1]	[7:4]	[3:0]
4	[7:5]	-	[7:5]	[4]	[7:5]	[4:3]	[7:5]	[4:2]	[7:5]	[4:1]	[7:5]	[4:0]
5	[7:6]	[5]	[7:6]	[5:4]	[7:6]	[5:3]	[7:6]	[5:2]	[7:6]	[5:1]	[7:6]	[5:0]
6	[7]	[6:5]	[7]	[6:4]	[7]	[6:3]	[7]	[6:2]	[7]	[6:1]	[7]	[6:0]
7	-	[7:5]	-	[7:4]	-	[7:3]	-	[7:2]	-	[7:1]	-	[7:0]

będzie ono oczekiwało do momentu, aż wszystkie ważniejsze zdarzenia zostaną wykonane. W przypadku, gdy wystąpią liczne zdarzenia o tym samym priorytecie wywłaszczania, jednostka NVIC obsłuży je w kolejności wykonującej z subpriorytetu. Zdarzenia o tych samych wartościach obu priorytetów będą wykonywane w kolejności wynikającej z numeracji źródeł z których pochodzą. Należy przy tym zaznaczyć, że wyższy priorytet reprezentowany jest przez mniejsze liczby, przy czy wartość 0 oznacza najwyższy priorytet.

Przykładowo, jeśli priorytety przerwań zostaną zadeklarowane tak, jak w tabeli 2 to:

- przerwanie IRQ0 będzie mogło wywłaszczyć zarówno IRQ1, jak i IRQ2,
- przerwania IRQ1 i IRQ2 będą oczekiwały, jeśli wystąpią w momencie obsługi IRQ0,
- jeśli zarówno IRQ1, jak i IRQ2 oczekują, IRQ1 zostanie wykonane przed IRQ2,
- jeśli IRQ1 wystąpi w trakcie wykonywania obsługi IRQ2, obsługa IRQ1 zostanie wstrzymana do momentu zakończenia procedury związanej z IRQ2.

W tabeli 3. przedstawiono sposób podziału bitów priorytetu pomiędzy priorytet główny i subpriorytet, w zależności od wielkości rejestru priorytetów oraz ustawienia grup priorytetów. Przykładowo, jeśli rejestr priorytetów ma rozmiar czterech bitów, a grupom priorytetów nadano wartość 4, oznacza to, że bity [7:5] będą reprezentować priorytet główny, a bit [4] będzie odpowiadać subpriorytetowi. Pozostałe bity [3:0] nie będą używane. W tej sytuacji zdarzenie o priorytecie 0x20 będzie wywłaszczało zdarzenie o priorytecie 0x40. W przypadku gdy zdarzenia o priorytetach 0x40 i 0x50 wystąpią w trakcie obsługi zdarzenia o priorytecie 0x20, będą oczekiwały na swoją kolej, przy czym to o priorytecie 0x40 zostanie wykonane przed 0x50.

W przypadku, gdy konstruktor nie umie zdecydować, jaką ustawić grupy priorytetów, zaleca się przypisanie jej wartości 0. Warto przy tym zaznaczyć, że w przypadku 8-bitowego rejestru priorytetów, uzyska się w ten sposób maksymalnie 128 poziomów priorytetów głównych. Kod ustawiający wartość grup priorytetów w NVIC został przedstawiony na listingu 5.

Priorytety przerwań bloków peryferyjnych konfigurowane są w oparciu o ich numery IRQ, tak jak na listingu 6.

Mapowanie pamięci RAM

Wielu producentów układów z rdzeniami ARM7TDMI często wbudowuje w kontrolery pamięci mechanizm, który pozwala na określenie, który z dostępnych bloków pamięci (ulotnej lub nieulotnej) będzie widoczny pod adresem wskazywanym przez wektor zerowania (zazwyczaj jest to adres 0x0). Pozwala to na uruchomienie aplikacji z określonym zestawem wektorów w pamięci nieulotnej, a następnie na przełączenie się do innego zestawu wektorów, umieszczonego w początkowym obszarze pamięci RAM.

Listing 5. Kod, za pomocą którego możliwe jest ustawienie wartości grup priorytetów w NVIC

```
#define NVIC_AIRCR  (*((unsigned int *)0xE000ED0C))
NVIC_AIRCR = (0x05fa << 16) |   /* klucz dostępu */
(0 << 8);          /* grupa priorytetów 0 */
```

Listing 6. Konfiguracja priorytetów przerwań w układach z Cortex-M3

```
#define NVIC_IPR  ((unsigned char *)0xE000E400)
#define NVIC_ISER ((unsigned int *)0xE000E100)
#define NVIC_ICER ((unsigned int *)0xE000E180)
#define NVIC_ICPR ((unsigned int *)0xE000E280)
/* wyłącz obsługę IRQn */
NVIC_ICER[IRQn >> 5] = 1 << (IRQn & 0x1f);
/* ustaw priorytet IRQn */
NVIC_IPR[IRQn] = priority;
/* wyczyść oczekujące IRQn */
NVIC_ICPR[IRQn >> 5] = 1 << (IRQn & 0x1f);
/* włącz obsługę IRQn */
NVIC_ISER[IRQn >> 5] = 1 << (IRQn & 0x1f);
```

W przypadku rdzenia Cortex-M3, tego typu kontrolery pamięci nie są potrzebne, gdyż jednostka NVIC pozwala na umieszczanie tablicy wektorów obsługi przerwań w praktycznie dowolnym miejscu pamięci. Aby przełączyć się pomiędzy jedną a drugą tablicą wektorów wystarczy wpisać do odpowiedniego rejestru offset, pod jakim jest ona umieszczona (listing 7).

Jedynym wymaganiem co do lokalizacji tablicy wektorów przerwań jest by była ona umieszczona pod okrągłym adresem (w sensie bitowym). Adres ten powinien być zaokrąglony do najbliższej potęgi dwójki ponad wielkość tablicy. Przykładowo, jeśli w układzie zastosowano 30 źródeł przerwań, to po uwzględnieniu 16 zdarzeń systemowych, na tablicę składa się 46 pozycji. Najbliższą większą potęgą dwójki jest 64, co w przypadku 4-bitowych słów w tablicy daje wartość 256 bajtów. Do takiej właśnie liczby należy zaokrąglać w tym przypadku adres, pod którym rozpoczyna się tablica z wektorami.

Przerwania szybkie (FIQ)

W rdzeniach Cortex-M3 nie zaimplementowano bezpośredniego odpowiednika przerwań szybkich (FIQ) stosowanych w ARM7TDMI. O ile przerwania niemaskowalne (NMI) mogą je przypominać, to nie pozwalają ładować danych do rejestrów – „cieni". Dlatego przenosząc przerwania FIQ z ARM7 do Cortex-M3 należy je zamienić na zwykłe przerwania o wysokich priorytetach.

Wykonywanie kodu z RAM-u

Popularną metodą zwiększania szybkości przetwarzani kodu krytycznego czasowo w układach ARM7TDMI jest wykonywanie go z wewnętrznej pamięci SRAM. Najczęściej jest on do niej automatycznie kopiowany, o ile zostanie oznaczony dyrektywą „ramfunc". Rozwiązanie to nie sprawdzi się w Cortex-ach, gdyż zostały one wykonane tak, by zmaksymalizować szybkość wykonywania poleceń pamięci przeznaczonej specjalnie na kod programu – tj. zazwyczaj z wbudowanej pamięci Flash. W przeciwieństwie do ARM-ów 7, odczyt kodu z RAMu będzie trwał dłużej, a nie krócej, gdyż ta sama szyna systemowa będzie użyta zarówno do przesyłania poleceń jak i danych. Z tego samego względu warto utrzymywać wspomnianą wcześniej tablicę wektorów obsługi wyjątków w pamięci kodu, a nie w RAM-ie.

Tłumaczenie kodu asemblera

W świecie procesorów 32-bitowych, ręcznie tworzony kod asemblera jest w praktyce używany praktycznie tylko w sytuacjach, gdy danych

Listing 7. Przełączenie z jednej tablicy wektorów przerwań na inną, umieszczoną w dowolnym obszarze pamięci.

```
void *new_vector_table[] = {
    0,
    new_System_Init,
    new_NMI_Handler,
    new_HardFault_Handler,
    new_MemManage_Handler,
    new_BusFault_Handler,
    new_UsageFault_Handler,
    /*
     * pozostałe wskaźniki obsługi (w tym wskaźniki obsługi
     * przerwań pochodzących od bloków peryferyjnych)
     */
};
#define SCB_VTOR  (*((unsigned int *)0xE000ED08))
unsigned int pv = (unsigned int)new_vector_table;
if (pv < 0x20000000) {
    /* tablica wektorów jest w pamięci z kodem programu */
    SCB_VTOR = pv;
}
else {
/* tablica wektorów jest w pamięci SRAM - liczba 29 oznacza,
 * że tablica jest w pamięci SRAM
 */
    SCB_VTOR = (pv - 0x20000000) | (1 << 29);
}
```

Listing 8. Warunkowe wyczyszczenie 8 bajtów z adresu z rejestru R2 lub R3 w zależności od wartości zapisanej w R1 za pomocą zestawu instrukcji rdzenia Cortex-M3

```
cmp   r1, #0
ittee eq
streq r0, [r2, #0]   ; jeśli r1=0 zapisz 4 pierwsze bajty r2
streq r0, [r2, #4]   ; jeśli r1=0 zapisz 4 kolejne bajty r2
strne r0, [r3, #0]   ; jeśli r1!=0 zapisz 4 pierwsze bajty r3
strne r0, [r3, #4]   ; jeśli r1!=0 zapisz 4 kolejne bajty r3
```

operacji nie da się wykonać w języku wyższego poziomu lub gdy byłyby one wykonane zbyt wolno. Rdzeń Cortex-M3 został skonstruowany tak, by ograniczyć konieczność stosowania asemblera, a w przypadkach gdy nadal jest on niezbędny, by mógł być używany bezpośrednio wewnątrz funkcji języka C.

Kod asemblera, w którym wykorzystano zestaw instrukcji ARM będzie musiał być przepisany do języka wyższego poziomu lub ręcznie przetłumaczony na instrukcje Cortex-M3 Thumb/Thumb-2. W praktyce nowoczesne kompilatory tworzą równie dobry kod z języka C, jak z pisanego w asemblerze, dlatego zaleca się stosowanie pierwszego z nich, co zapewnia większą przenośność kodu.

Jedną z korzystnych i często wykorzystywanych cech zestawu instrukcji ARM jest warunkowe wykonywanie poleceń, dzięki któremu można uniknąć skoków, za pomocą których pomija się określony fragment kodu. Podobną funkcję można znaleźć w zestawie instrukcji Cortex-ów M3. Za pomocą polecenia „IT" (if-then) możliwe jest warunkowe wykonanie od jednej do czterech następujących po sobie instrukcji, w zależności od wyniku zadanego porównania. Użycie instrukcji wykonania warunkowego typowej dla Cortex-M3 zostało przedstawione na listingu 8. Od analogicznego kodu dla rdzenia ARM różni się tylko wstawieniem linijki ittee eq.

Cześć z instrukcji może być zakodowana zarówno jako 16-bitowe polecenia Thumb lub 32-bitowe Thumb-2. Wyboru sposobu kodowania można dokonać dopisując sufiks „.n" dla poleceń 16-bitowych lub „.w" dla 32-bitowych. Jeśli sufiks nie zostanie podany, asembler zazwyczaj zakoduje polecenie jako instrukcję Thumb.

Listing 9. Ustawianie i zerowanie obsługi przerwań w układach z rdzeniem ARM7TDMI

```
disable_irq:
    mrs   r0, CPSR_c      ; odczytaj CPSR
    orr   r0, r0, #0x80   ; ustaw bit I
    msr   CPSR_c, r0      ; zapisz zmodyfikowany CPSR
enable_irq:
    mrs   r0, CPSR_c      ; odczytaj CPSR
    bic   r0, r0, #0x80   ; wyczyść bit I
    msr   CPSR_c, r0      ; zapisz zmodyfikowany CPSR
```

Listing 10. Ustawianie i zerowanie obsługi przerwań w układach z rdzeniem Cortex-M3

```
disable_irq:
    mov   r0, #1
    msr   PRIMASK, r0
enable_irq:
    mov   r0, #0
    msr   PRIMASK, r0
```

Wyłączanie obsługi przerwań

Od czasu do czasu może zajść potrzeba, by wyłączyć obsługę przerwań. W przypadku układów z ARM7TDMI, często można tego dokonać za pomocą odpowiedniego rejestru kontrolera przerwań lub poprzez ustawienie bitu I rejestru CPSR procesora (listing 9). W Cortex-ach M3 specjalny rejestr PRIMASK pozwala wyłączyć wszystkie przerwania za wyjątkiem przerwań niemaskowanych oraz wyjątków spowodowanych błędami (listing 10).

Przerwania programowe

Układy z rdzeniami ARM7TDMI pozwalają na generowanie wyjątków poprzez instrukcję SWI. Jest ona używana głównie do komunikacji ze sterownikami lub innym uprzywilejowanym kodem, który nie może być wywołany bezpośrednio. W odpowiedzi na polecenie SWI, w rejestrze R14 pojawia się adres instrukcji następującej po SWI, procesor wyłącza obsługę przerwań, zmienia tryb pracy na SVC i skacze pod adres znajdujący się w wektorze obsługi przerwania SWI. Dalsza obsługa przerwania może zostać zaprogramowana w sposób przedstawiony na listingu 11.

Obsługa przerwań programowych w rdzeniu Cortex-M3 odbywa się nieco inaczej, gdyż NVIC automatycznie zrzuca na stos rejestry R0-R3, R12, LR, PC oraz PSR. Procedura obsługi musi najpierw określić, który ze stosów został użyty, by dostać się do przekazywanych parametrów (list 12).

Kolejną różnicą, jaką należy wziąć pod uwagę przy przenoszeniu kodu jest fakt, że o ile procedura obsługi wyjątku SWI może wywołać kolejny wyjątek SWI, to w rdzeniach Cortex-M3 NVIC zgłosi błąd przy próbie wywołania wyjątku o tym samym priorytecie co obsługiwany.

Obsługa błędów

Zarówno rdzeń ARM7TDMI, jak i Cortex-M3 mają specjalne zdefiniowane specjalne wyjątki, które zostaną wywołane w momencie gdy powstanie problem związany z dostępem do pamięci lub przetwarzaniem instrukcji. Ich wystąpienie zazwyczaj oznacza usterkę sprzętową lub błąd w oprogramowaniu i w związku z tym ich obsługa sprowadza się zazwyczaj do wstrzymania dalszej pracy procesora.

Ważnym elementem radzenia sobie z takimi błędami jest zapisanie adresu instrukcji, która była wykonywana w momencie wystąpienia błędu. Wraz z deasemblacją kodu, zawartością rejestrów procesora i fragmentem stosu powinny wystarczyć do określenia przyczyny problemu. W układach z ARM7TDMI wykonywane polecenie może zostać odnalezione poprzez odjęcie 4 od wartości rejestru LR w przypadku błędów instrukcji lub poprzez odjęcie 8 w przypadku błędów związanych z danymi. W Cortexach-M3 licznik programu jest wrzucany na stos w momencie wystąpienia błędu i może być pobrany za pomocą kodu przedstawionego na listingu 13.

W Cortexach-M3 mogą wystąpić następujące rodzaj wyjątków spowodowanych błędami:

- Błąd użytkowania (Usage fault) – spowodowany nieznanymi instrukcjami lub niewyrównanym adresem dostępu do kodu
- Błąd pamięci (Memory management fault) – spowodowany próbą dostępu do nieuprzywilejowanego obszaru pamięci
- Błąd szyny (Bus fault) – spowodowany próbą dostępu do błędnych obszarów pamięci
- Twardy błąd (Hard fault) – uruchamiany w przypadku, gdy pozostałe trzy wyjątki nie mogą zostać wywołane

Listing 11. Sposób obsługi przerwania programowego za pomocą zestawu instrukcji ARM:

```
swi_exception_handler:
    stmfd sp!, {r10}
    ldr   r10, {r14, #-4}     ; pobierz instrukcję SWI
    bic   r10, r10, #ff000000 ; pobierz parametr instrukcji
                              ; SWI
        ; wykonaj kod zależny od parametru z rejestru r10
    ldmia sp!, {r10, pc}^     ; wróć do poprzednio
                              ; wykonywanego kodu
```

Listing 12. Sposób obsługi przerwania programowego w rdzeniu Cortex-M3
```
svc_exception_handler:
    tst   lr, #4
    ite   eq
    mrseq r0, MSP
    mrsne r0, PSP
    ldr   r1, [r0, #24]
    ldrb  r1, [r1, #-2]       ; pobierz parametr SVC
        ; wykonaj kod zależny od parametru z rejestru r1
    bx    lr                  ; wróć do poprzednio
                              ; wykonywanego kodu
```

Listing 13. Sposób pobrania adresu wykonywanej instrukcji w przypadku wystąpienia błędu w rdzeniu Cortex-M3
```
    tst   lr, #4
    ite   eq
    mrseq r0, MSP
    mrsne r0, PSP
; sprawdź czy r0 zawiera poprawną wartość
    tst   r0, #3           ; by uniknąć drugiego błędu

    bne   skip
    ldr   r1, =STACK_ADDRESS_MIN
    cmp   r0, r1
    bmi   skip
    ldr   r1, =STACK_ADDRESS_MAX - 32
    cmp   r0, r1
    bpl   skip
    ldr   r1, [r0, #24]       ; r1 <= PC
skip:
```

Listing 14. Kod, za pomocą którego odczytywane są wartości rejestrów błędu
```
    ldr   r0, =0xE000ED28
    ldr   r1, [r0, #0]   ; r1 <= CFSR
    tst   r1, #0x80      ; MMARVALID==1?
    it    ne
    ldrne r2, [r0, #12]  ; r2 <= MMAR
    tst   r1, #0x8000    ; BFARVALID==1?
    it    ne
    ldrne r2, [r0, #16]  ; r2 <= BFAR
```

Listing 15. Użycie instrukcji LDREX i STREX rdzenia Cortex-M3 do obsługi semaforów
```
take_semaphore:
    ldr   r0, =semaphore_addr
    ldrex r1, [r0]
    cbnz  r1, take_semaphore   ; semafor jest zajęty
                               ; spróbuj ponownie
    mov   r1, #1
    strex r2, r1, [r0]         ; spróbuj ustawić semafor
    cbnz  r2, take_semaphore   ; semafor jest jednak zajęty
                               ; spróbuj ponownie
give_semaphore:
    ldr   r0, =semaphore_addr
    mov   r1, #0
    str   r1, [r0]             ; wyzeruj semafor
```

Listing 16. Przykład kodu sterującego wyprowadzeniem PIO
```
ldr   r0, =pio_set_reg
ldr   r1, =pio_clear_reg
mov   r2, #1
str   r2, [r0]   ; ustaw PIO
str   r2, [r1]   ; zeruj PIO
```

Wyjątki 1-3 mają priorytety konfigurowane przez użytkownika i muszą być uaktywnione, by mogły być wywołane. W sytuacji, gdy wystąpi błąd, który nie będzie mógł być obsłużony np. przez nieaktywny wyjątek 1-3 lub będzie miał zbyt niski priorytet, procesor wywoła 4. wyjątek.

Ponadto układy z Cortex-M3 mają kilka rejestrów, które ułatwiają określenie warunków, w jakich zaszedł problem. Noszą one nazwy:
- Usage Fault Status Register (UFSR)
- MemManage Fault Status Register (MMSR)
- Bus Fault Status Register (BFSR)
- MemManage Fault Address Register (MMAR)
- Bus Fault Address Register (BFAR)

Trzy rejestry stanu (UFSR, MMSR and BFSR) mogą być odczytane w całości jako jeden 32-bitowy rejestr CFSR (Combined Fault Status Register). Rejestry MMAR i BFAR zawierają adres, który spowodował odpowiadający im błąd, o ile bit MMARVALID lub BFARVALID jest ustawiony. Aby je odczytać należy zastosować kod z listingu 14.

Niektóre błędy związane z szyną danych mogą wystąpić dopiero po wykonaniu szeregu instrukcji po tej, która spowodowała problem. Jeśli tak się stanie, bit IMPRECISERR rejestru BFSR zostanie ustawiony.

Warto też dodać, że instrukcja SVC nie może być użyta w trakcie obsługi błędu „Hard fault", gdy zawsze będzie ona miała niższy priorytet. Próba jej użycia wywoła w tej sytuacji drugi błąd tego samego typu. Gdy tak się stanie, Cortex-M3 blokuje się, do momentu wystąpienia przerwania NMI, użycia debugera lub zerowania mikrokontrolera.

Instrukcja SWP

W zestawie instrukcji rdzenia ARM7TDMI znajduje się instrukcja SWP, która nie ma swojego bezpośredniego odpowiednika w zbiorze poleceń rdzenia Cortex-M3. W praktyce, SWP stosuje się głównie do obsługi semaforów, gdyż pozwala ona na jednostkowy odczyt a następnie zapis do pamięci.

Przy przenoszeniu kodu na nowe układy analogiczne funkcje można uzyskać poprzez zastosowanie instrukcji LDREX i STREX, tak jak zostało to przedstawione na listingu 15.

Kontrola czasu wykonywania

Rdzeń Cortex-M3 został tak zaprojektowany, by instrukcje LDR i STR przetwarzać potokowe, gdy tylko jest to możliwe. Dzięki temu następujące po nich instrukcje zaczynają się wykonywać zanim te pierwsze skończą działanie. W normalnej sytuacji technika ta zwiększa szybkość przetwarzania kodu, ale może też spowodować pewne problemy w aplikacjach, w których precyzyjnie zarządzano potokiem wykonywanych poleceń.

Przykładowo, kod z listingu 16 generuje impuls na wyprowadzeniu PIO poprzez zapis danych do odpowiednich rejestrów. W przypadku układów Cortex-M3, kod ten wytworzy impuls długości dwóch cykli zegarowych, co wynika z czasu wykonania drugiej instrukcji STR. Sensownym założeniem będzie przyjęcie, że dodanie instrukcji NOP pomiędzy pierwszy a drugi zapis wydłuży ten impuls o jeden cykl, ale w praktyce pozostanie on tak samo długi jak wcześniej. Dzieje się tak, gdyż instrukcja NOP zostanie wykonana w trakcie drugiego cyklu instrukcji STR. Dopiero dodanie drugiej instrukcji NOP wydłuży długość impulsu o jeden cykl.

Optymalizacja

Po zakończeniu wstępnego przenoszenia kodu warto sięgnąć do nowych instrukcji Cortex-ów, które pozwalają na zoptymalizowanie programu i zwiększenie wydajności.

Przykładowo, rdzenie Cortex-M3 pozwalają na wygodny dostęp do dowolnych fragmentów pamięci, co w przypadku zestawu instrukcji ARM nie było możliwe. W starszych układach zmiana pojedynczego bitu w pamięci wymagała pobrania całego bloku pamięci do rejestru, wykonania operacji bitowych na rejestrze, a następnie zapisania całego rejestru z powrotem do pamięci. Jest to o tyle problematyczne, że jeśli wykonywanie wątku zostanie wstrzymane przez przerwanie, które zapisuje do tego samego obszaru pamięci, po powrocie do wykonywania wątku dane w tej pamięci zostaną uszkodzone poprzez nadpisanie ich nieaktualnymi informacjami. By mieć pewność, że operacja ta zawsze się powiedzie, konieczne byłoby wyłączenie na jej czas obsługi przerwań, a następnie ich ponownie odblokowanie. Oznacza to, że zapis pojedynczego bitu wymaga przynajmniej 5 operacji.

W układach z Cortex-M3 wprowadzono mechanizm, który pozwala modyfikować pojedyncze bity w pierwszym 1 MB pamięci odnosząc Siudo nich jako do pojedynczych słów z określonych 32 MB pamięci SRAM. Przykładowo, zapisanie jedynki pod adres 0x22000000 ustawi bit 0 słowa o adresie 0x20000000 (listing 17).

Todd Hixon, Atmel

Listing 17. Dostęp do pojedynczych bitów pamięci w układach z Cortex-M3
```
static unsigned int x;
unsigned int *p = (unsigned int *)(((unsigned int)&x & 0xf0000000) +
            0x02000000 + (((unsigned int)&x & 0x000ffffc) * 32));
x = 0;
p[0] = 1;              /* ustaw bit 0 w x */
p[1] = 1;              /* ustaw bit 1 w x */
p[31] = 1;             /* ustaw bit 31 w x */
if (p[0]) p[30] = 1;   /* ustaw bit 30 w x jeśli bit 0 jest
ustawiony */
/* x wynosi teraz 0xc0000003 */
```